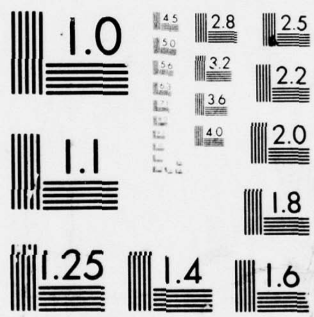
AD-A044 201 RAND CORP SANTA MONICA CALIF F/G 17/2
INTERPROCESS COMMUNICATION EXTENSIONS FOR THE UNIX OPERATING SY--ETC(U)
JUN 77 S ZUCKER F49620-77-C-0023
UNCLASSIFIED R-2064/2-AF NL

| OF |
AD
A044201

END
DATE
FILMED
10-77
DDC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

6

R-2064/2-AF
June 1977

Interprocess Communication Extensions for the UNIX Operating System: II. Implementation

Steven Zucker

A Project AIR FORCE report
prepared for the
United States Air Force

Rand
SANTA MONICA, CA. 90406

The research reported here was sponsored by the Directorate of Operational Requirements, Deputy Chief of Staff/Research and Development, Hq. USAF under Contract F49620-77-C-0023. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

Published by The Rand Corporation

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: (14) R-2064/2-AF	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): (6) Interprocess Communication Extensions for the UNIX Operating System: II. Implementation	5. TYPE OF REPORT & PERIOD COVERED: (9) Interim rept.
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): (10) Steven Zucker	8. CONTRACT OR GRANT NUMBER(s): (15) F49620-77-C-0023
9. PERFORMING ORGANIZATION NAME AND ADDRESS: The Rand Corporation, 1700 Main Street, Santa Monica, Ca. 90406	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Project AIR FORCE Office (AF/RDQA), Directorate of Operational Requirements, Hq USAF, Washington, D.C. 20330 (11)	12. REPORT DATE: June 1977
	13. NUMBER OF PAGES: 19 (12) 24p.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

No restrictions

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Data Processing
Data Transmission
Computer Systems Programs
Coding Theory
Software (computers)
Operating Systems (Computers)
Interprocess Communication

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

see reverse side

296 600 LB

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

The UNIX operating system for the PDP-11 series of minicomputers has gained wide popularity in academic and government circles. This report considers inter-process communication (IPC) facilities with the goal of developing an improved IPC capability for UNIX. A companion report outlines the major issues involved in providing IPC, describes the standard UNIX IPC facilities, and points out several of their weaknesses. The present report describes the "port" mechanism developed at Rand to overcome some of those weaknesses. It presents details of the implementation as well as sufficient background material to enable the UNIX programmer to understand how ports work and how to use them. (See R-2064/1-AF.) (PB)

R-2064/2-AF
June 1977

Interprocess Communication Extensions for the UNIX Operating System: II. Implementation

Steven Zucker

A Project AIR FORCE report
prepared for the
United States Air Force

Rand
SANTA MONICA, CA. 90406

PREFACE

The UNIX operating system for the PDP-11 series of minicomputers has gained wide popularity in academic and government circles. Under the Project AIR FORCE (formerly Project RAND) study effort, "Information Sciences Research," The Rand Corporation is engaged in analyzing, evaluating, and developing computer operating system concepts with UNIX. Recent work has dealt with such topics as security, file systems, performance, user interfaces, network access, and office automation.

This report, together with its companion report, R-2064/1-AF,* describes the current state of work in the area of interprocess communication (IPC). The companion report outlines the major issues involved in providing IPC, describes the standard UNIX IPC facilities, and points out several of their weaknesses. The present report describes the "port" mechanism developed at Rand to overcome some of those weaknesses. It presents details of the implementation as well as sufficient background material to enable the UNIX programmer to understand how ports work and how to use them. While the report is intended principally as a user's and implementer's guide, the reader with a general knowledge of operating systems should be able to follow the discussion without difficulty.

---

* Carl Sunshine, Interprocess Communication Extensions for the UNIX Operation System: I. Design Considerations, The Rand Corporation, R-2064/1-AF, June 1977.

Preceding Page Blank - Not Filmed

SUMMARY

The interprocess communication (IPC) facilities in the UNIX operating system, known as pipes, suffer from several deficiencies. They do not allow communication between processes unless prior arrangements have been made by a common ancestor of the processes desiring to communicate. In addition, they do not support the notion of a "message"; all the data written on a pipe are merged into a single stream. Thus, it is impossible for a receiving process to determine which of several potential senders is the source of the data received, or even the boundaries between the data coming from different senders. These deficiencies and others have led to the development of a new IPC mechanism for UNIX, known as a port.

This report describes the implementation of ports as UNIX "special files." Ports build on the existing pipe mechanism to achieve system buffering of messages and use the UNIX directory structure to provide a naming capability so that unrelated processes can communicate. The ordinary UNIX access protection mechanisms also carry over to ports. Thus, ports constitute a logical extension to the UNIX file system and have proved quite easy to implement.

In addition to the implementation description, this report contains all the information required to use ports in programs. The behavior of the UNIX input/output operations as they apply to ports is described in Sec. IV. Details of the system call that creates ports are provided in an appendix.

Preceding Page Blank - Not Filmed

CONTENTS

Preceding Page Blank - NOT FILMED

I. INTRODUCTION

This is the second of two companion reports dealing with interprocess communication (IPC) in the UNIX operating system. The first report [1] presents an overview of the IPC mechanisms that have appeared in various systems and analyzes the weaknesses of IPC in UNIX. The present report describes a mechanism, called a port, developed at Rand to eliminate some of those weaknesses.

The port mechanism described here provides a means by which arbitrary UNIX processes can communicate in stream- or message-oriented fashion. Much of the needed background is provided in Sec. II, where the UNIX pipe facility on which ports are based is described. In Sec. III, the motivation for ports is presented and the features they provide are described. Section IV provides a more thorough description of the implementation of ports and the details regarding their use, and in Sec. V their security-related aspects are summarized. Details of the system call that creates ports are presented in an appendix.

II. PIPES

Since this report deals with the specific measures taken to improve the UNIX operating system, the reader is presumed to have some familiarity with UNIX. "The UNIX Time-Sharing System" [2] will provide most of the the requisite background. While a user-(programmer)-level view is sufficient for understanding the essential features of the IPC mechanism described here, some of the details require knowledge of the internal operation of the UNIX kernel. Of particular importance is an understanding of the UNIX pipe construct.

A UNIX pipe is an unnamed file that is used for stream-oriented data transfer between related processes. A process creates a pipe by invoking a system-provided primitive function; when the process forks, its child or children inherit file descriptors (process-local identifiers by which files are referenced), for the pipe that they, as well as the parent, can read or write. The only differences between pipes and ordinary files are:

1. They are created by means of a special system call that returns two file descriptors--one for reading and another for writing. Thus the process that creates a pipe can pass on to its descendants capabilities for reading or writing only, or capabilities for both.

2. They have no names and are deleted when closed by all processes having file descriptors for them (all files are implicitly closed on process termination).

3. There is implicit synchronization that prevents the writers of a pipe from getting more than a fixed number of characters ahead of its reader(s) by blocking the writers (putting them to sleep) until the reader catches up. When the reader catches up, the file representing the pipe is truncated to zero length.

Pipes are buffered on three levels:

1. The data are buffered in kernel memory. The buffer space is that for ordinary disk files and requires no additional space in the kernel.

2. As blocks are needed by the rest of the system, kernel buffers blocks are written to disk storage. Again, the transfer of blocks to and from disk is handled just as it is for ordinary files.

3. If the processes writing a pipe get too far ahead of the reading processes, they are suspended until the readers catch up. This is a kind of "buffering in time."

The new mechanism augments pipes by providing two new capabilities:

1. It enables a reader to demultiplex the input stream into "messages" and to determine the source process that wrote each message.

2. It provides a naming capability so that unrelated processes can communicate with each other.

In most other respects, the new mechanism is practically identical to a pipe. It shares the buffering mechanisms and therefore makes use of much of the existing kernel code. Thus, building on pipes has led to considerable economy of implementation. In addition, since pipes are a familiar construct to UNIX programmers, the extensions are considerably easier for users to understand and use than mechanisms that require new, specialized system calls. The new mechanism is called a port.

III. PORTS

Ports were originally conceived to solve the problem of multiple asynchronous inputs to a process. This problem arises whenever a UNIX process receives inputs from two or more sources and wishes to wait for the first input from either of them. In UNIX a read of any one file (or pipe) causes the reading process to go to sleep until data are available from that file or until the reader is signaled, in which case data may be lost. This motivated an earlier Rand addition to UNIX, the "empty" system call, which tells a process whether or not data are available for reading from a particular pipe or terminal. While useful in some contexts, the "empty" call is often unacceptable because it requires a process to poll its inputs, thus placing a constant demand on the processor.

The port mechanism solves the multiple input problem by making it possible to multiplex all of the various inputs into a single stream. A port is created using a system call that is similar to the pipe system call, except that the port is marked for special treatment on writes. On a write to a port, the data to be written are preceded by a header containing information identifying the writer and a character count. The writer ID may consist of the process number of the writing process, the user ID number of the owner of that process, a system-assigned "process group identifier" (that is the same for all processes sharing a single "open" of the port), or any combination (including none) of the above. The UNIX kernel supples the header, so that: (1) from the point of view of the writer, writes to a port are no different than writes to any other device (writes are

"device-independent"), and (2) the reading process can tell unequivocally who did the writing and can parse the data into the proper units, since headers cannot be "forged."

A port may be given a name and write permissions at the time it is created. The name and protection information is specified just as for an ordinary file and is implemented using the already existing mechanisms within UNIX, thus providing economy of implementation as well as general harmony with the rest of the system. Since ports may be opened for writing in the same way as ordinary files by users with write permission, they make possible interprocess communication between unrelated as well as related processes.

With respect to the reader, a port can be created (and implicitly opened), read, and closed like any other file. A port is an exclusive use "device" with respect to reading. Creating a port implicitly opens it for reading, and only the creator has read access to it. Note that this restriction does not rule out reading of the port by descendants of the process that created it. They may be passed a file descriptor for the already opened port, but it is assumed that they will be cooperative in its use. Cooperation involves synchronization by external means.* Assuming synchronization, the reading processes will be regarded as a single entity referred to below as the "reader."

There are two reasons for the "one reader" restriction on ports. First, unless individual messages could be directed to a particular

---

* An example of such synchronization is the use of the "wait" system call by the shell (the UNIX command language interpreter) to prevent simultaneous reading of input data from a terminal by more than one process.

reader, which can be achieved easily by the use of separate ports, having multiple readers would seem to serve no purpose. Second, if multiple readers were allowed, there would have to be some means of insuring that readers would always receive complete messages, i.e., that no message would be split among several readers. This would require that the system perform considerable bookkeeping, perhaps to the point of providing interlocks on simultaneous access. Instead, in the present implementation, once data have been written into a port with its header they are treated as stream data just as on a pipe; the system has only to pass the data on to the reader as called for without regard to contents or message boundaries. The above arguments, of course, apply only to the case of more than one simultaneous reader. If applications for multiple readers, disjoint in time, suggest themselves, the restriction could be removed.

In the simplest usage mode, a port can be regarded by its writer as an output file or "device" that can be opened, written, and closed like an ordinary file. The writer need not even be aware that the "file" is a port. Writing to a port, like writing to a pipe, can be regarded as writing to the process or group of related processs that are reading the pipe.

A write that would overflow the pipe that serves as a buffer for the port data is split into several portions, each with its own header. To facilitate message- as well as stream-oriented communication, there is an "end of message" indicator in the header to enable a reader to recognize logical message boundaries defined by the writer. A message in this context is the data presented to the port by a single UNIX "write" system call. The pieces of a "split" message

may be interspersed with those of other messages; it is the reader's responsibility to reconstruct whole messages based on the headers if it wishes to do so. While it would have been possible to guarantee that the data from each write would be contiguous, the decision to split messages was made for two reasons. First, splitting messages makes it less likely that a single writer will monopolize the port by writing many very long messages. Second, it proved to be somewhat easier to implement efficiently.

When all the processes in a "process group" (that is, all writers sharing a single "open" of the port) have closed it, a zero length message (i.e., just a header with an end-of-message flag and information identifying the writer) is placed into the port. Thus, the reader can determine when communication from a process group has ended.

IV. IMPLEMENTATION

Ports, as stressed earlier, make use of existing UNIX mechanisms for buffering and naming. This section describes the implementation in greater detail

The UNIX treatment of "special files" has made the implementation of ports particularly simple. A UNIX special file, like an ordinary disk file, is represented by an "inode." An inode is a data structure that contains the owner's ID, protection information (read, write, and execute permission indicators for the owner himself, others in the owner's group, and all other users), and other descriptive data. The directory structure provides one or more names by which to reference an inode, as well as additional protection from unauthorized reference. Within each special file inode there are two 8-bit numbers, a major device number and a minor device number. The major device number determines which driver routines are used to perform the opens, closes, reads, writes, and special functions (stty's) directed to the "file." The minor device number is a parameter that is passed to the driver when an operation is invoked and usually determines which one of several (up to 256) "devices" of that type is to be affected.

It has been relatively easy to implement ports as UNIX special files. Naming and access control are provided by the same mechanisms that apply to ordinary files. In-core and disk buffering of port data use the mechanisms mentioned above that already exist for pipes. Implementing ports as special files provides a very clean interface

to the code that manages port I/O. Except for the system call that creates ports and minor (one-line) changes to the "close" and "seek" system calls, all of the code required to implement ports is contained in the port "device" driver, which is installed in the kernel in the usual way. Thus, UNIX already provides the user interface through the usual I/O system calls.

The reason that a separate port creation mechanism (the port system call described in detail in the appendix) is desirable is that the creation of special file inodes is a privileged operation. While it would be possible to implement ports by providing a privileged process for port creation and management, investigation has shown this to be less convenient, efficient, and flexible than the new system call.

All the data structures necessary to support the IPC mechanisms were already present in UNIX, namely, the files table and inodes. A port uses two inodes. The first is a special file inode with a major device "port," the other is a disk file inode identical to those that the pipe call creates. The minor device for the first inode is set at the time of port creation to indicate what (if any) data are to be put into the headers prefixed to data written on the port. If a non-null name is given, the port call creates a directory entry for the inode. One of the seven spare "address" words in the port inode points to the disk file inode. The file table entry (entries, if opened for reading and writing) uses one of the several spare flag bits to indicate a "port." This enables the seek system call to return an error as it does for pipes, and the close system call to notify the driver when each process group, not just the last, closes the port.

The operations that can be performed on ports are given below, with their effects and restrictions on their use.

open: A port can be opened only for writing and then only if:

(1) it was given a name when it was created (otherwise it could not be specified as an argument to the open system call),

(2) the caller has write permission (this check is performed by UNIX before the driver open routine is called), and

(3) the inode is already opened for reading. Thus, one process can communicate with another only if the other has indicated (by creating a port leaving it open for reading) that it is willing to listen.

write: The driver uses the information in the port-type inode to locate the disk file inode, which it treats as a pipe. Headers are prefixed to the data if required. It may be necessary for the driver to split a single write into a number of "chunks," putting the writer to sleep until the reader catches up (i.e, makes more room on the "pipe" by reading). There were two possible conventions:

(1) Guarantee that all of the data from one UNIX "write" be written before any other writer could place data in the "pipe," or

(2) Split a single UNIX write into several "chunks," each having its own header, and intersperse the chunks from different writes as required.

Under (1) it would have been easier for one writer, through malice or error, to monopolize the use of the port by doing large (up to 65,535 byte) writes. Alternative (2), which was chosen, necessitated the use of the end-of-message flag to enable the reader to recognize the "write" boundaries (messages). The problem that this introduces is that the reader may receive many partial messages before any one message is completed, and has to buffer partial messages in user space when message boundaries are significant.

If a write is attempted after the reader has closed its end of the "pipe," a signal (#11) is generated, just as on a pipe.

read: Ports look like pipes with respect to reading. The read call returns the minimum of the number of characters specified and the number in the "disk file," sleeps if the file is empty, and returns an end-of-file indication when the last writer does a close. Ports provide stream data; it is the responsibility of the reader to keep track of message boundaries based on header information, an operation that can be performed easily by a user subroutine. Since the data are stream-oriented, the reader can read them in whatever manner it likes, reading headers and data with separate operations, or reading and buffering blocks for greater efficiency.

close: The close function behaves as it does on any other file, except that if headers are being written, a zero length message (header only, with end-of-message bit set) is written when each process group closes the file. This required only a one-line change in the UNIX "close" kernel code and enabled a reader to detect a logical end-of-file condition for each writer. Unfortunately, the last close of a port cannot make its directory entry disappear, since there is no storage available in which to keep its name. Furthermore, there is no way to control the proliferation of names for a given inode (port, in this case) through *link (ln)* operations. Thus, it is the user's responsibility to remove the names he creates (with the "unlink" system call or the "rm" (remove) command. However, while a careless user may use up inodes and directory entries, it is just as if he had failed to delete temporary files of zero length.

stty/gtty: The special function calls, stty and gtty, make it possible to implement "device-particular" operations in UNIX. At present, stty calls on ports are ignored, but the following paragraph indicates some of the options that could be implemented to enable the reader to tailor the characteristics of a port to meet special requirements.

Since reading a port, like reading any other UNIX file, causes the reader to block until data are available, it might be useful for a reader to be able to determine when and how many data are available. The stty call could be used to:

(1) Request that a signal be generated whenever data are written on an empty pipe, or

(2) Return the number of bytes available to be read.

In some applications it might be important for the reader of a port to exercise some control over the writers, especially to prevent some writers from monopolizing use of the port at the expense of others. The special function calls might be employed in this regard to:

1. Prescribe a maximum "chunk size," with the driver enforcing a rule that a given process not be allowed to write two consecutive chunks until the pipe is emptied by the reader.

2. Return the number of open write file descriptors and the number of writers waiting to write (number of "active" writers). This could be used to compute a reasonable chunk size for the above.

3. Cause a writer to receive an error indication instead of going to sleep if all of its data can not be accepted.

4. Set a maximum message size (for a single write) and cause writers to receive an error indication if they try to write more.

5. Suspend a particular writer (by putting it to sleep when it tries to write), or reactivate a suspended writer. Note that permission to suspend is implicit in the fact that the writer is using the port.

6. Reactivate all writers or a single selected writer.

V. SECURITY IMPLICATIONS

The creator of a port has complete control over it. When a process creates a port, the (effective) owner of the process becomes the owner of the port. The port creation system call specifies a set of write permissions for the creator himself, other users in his group, and all other users, just as the "creat" system call does for ordinary files. Only the specified classes of users plus the privileged "superuser" may open the port for writing. No process, even one owned by the "superuser," can open the port for reading, so the creator of the port, and those of its descendants to which it chooses to pass file descriptors, can read it without fear of losing data to other processes.

Since the kernel supplies the headers on port data and headers include a byte count, no writer can "forge" a header. Thus the reader can rely on knowing what process, process group, and user (real and effective) wrote each byte that it receives.

Port names are directory entries, just as are the names of ordinary files. The port may be linked to (given other names) and may have its names removed by any user having permission to write in the relevant directories. Thus the "superuser" may remove the name for any port. Removing the name does not, however, prevent the reader and writers who have the port open from proceeding; it merely keeps other potential writers from gaining access to it.

A (possibly malicious) writer may write to the port as much and as often as desired in the absence of the flow control features mentioned at the end of the previous section. However, all the active

writers are suspended (blocked) when the combined data from the writers as a whole get ahead of the reader by a fixed number of bytes. When the reader catches up, all the blocked writers are reactivated at the same priority, so each then has an equal chance to gain access to the port. Thus, while any writer can delay service, none can completely deny access to the others. In addition, of course, the reader can ignore data from a writer that "misbehaves."

VI. REMARKS

The IPC mechanism for UNIX described here provides several capabilities that either could not be obtained, or could be obtained only unreliably or inefficiently, within standard UNIX. It should be noted, however, that the mechanism is directly concerned only with data transfer rather than synchronization. Improved process synchronization will probably have to rely on improvements in the UNIX signaling mechanism or development of an alternative mechanism.

Preceding Page Blank - Filmed

Appendix

DETAILS OF THE PORT SYSTEM CALL

The IPC mechanism described above requires only one new system call. The new call creates a port and opens it for reading (and, optionally, for writing also). The format of the call is as follows:

```
char *name;
int fdr, mode, fds[2];

fdr = port(name, mode, fds);
```

fdr: A returned file descriptor for reading, or -1 if the call fails.

name: The name by which a port may be opened for writing. It may be a pointer to a null string ("") in which case the port cannot be opened by other processes. The name may be a complete path name (beginning with a "/") or may be relative to the current directory, just as for the "creat" system call.

mode: This argument is a 16-bit word formed from the sum of the following:

0100000: If the "file" is to be opened for reading only, and not for writing (default is to open for both reading and writing).

040000: If the header supplied on writes should contain the process ID of the writer.

020000: If the header supplied on writes should contain the real user ID (in the low-order byte) and the effective user ID (in the high-order byte) of the writing process.

010000: If the header supplied on writes should contain the process group to which the writer belongs.

If none of the above three options is chosen, no header is supplied and the port reads just like an ordinary UNIX pipe. Otherwise, the indicated words are written in the above order,

followed by a word containing the character count of the data to follow and, in the high-order byte, a 1 if the last byte of the data is the last byte written by a single write call, indicating an "end of message."

0200: Write permission for creator.

020: Write permissions for others in the creator's group.

02: Write permissions for all other users.

If the file name is null the permission bits are not used, and the file is opened for reading and writing regardless of the "open flag" (0100000) setting.

fds: The file descriptors for reading and writing are returned in fds[0] and fds[1], respectively (fds[0] = fdr). If mode has the 0100000 component, the fds argument is not used.

Named ports can be opened for writing by processes with write permission, just as an ordinary UNIX file. A writing process need not be aware that it is writing a port, as the headers are supplied by the port "driver" routine in the kernel.

REFERENCES

1. Sunshine, Carl, Interprocess Communication Extensions for the UNIX Operating System: I. Design Considerations, The Rand Corporation, R-2064/1-AF, June 1977.

2. Ritchie, D. M., and K. Thompson, "The UNIX Time-Sharing System," Comm. ACM 17, 7, July 1974, pp. 365-375.